GD2S03
Advanced Software Engineering & Programming for Games

Bachelor of Software Engineering(BSE)
Game Development

- Overview
  - Functions
  - Closures
  - Enumerations
  - Classes and structures

Function

• Definition and calling a function
- Takes string as an input and return a string as output.
	`func greet(person: String) -> String {`
	`    let greeting = "Hello, " + person + "!"`
	`    return greeting`
	`}`
	`print(greet(person: "Anna"))`

• Functions Without Parameters	`func sayHelloWorld() -> String {return "hello, world"}`

• Functions With Multiple Parameters	`func greet(person: String, alreadyGreeted: Bool) -> String {`
	`    if alreadyGreeted{ return("Hello again" + person)}`
	`    else{ return greet(person:person)}`
	`}`

• Functions Without Return Values	`func greet(person: String) { }`

• Functions with Multiple Return Values - return as tupule	`func minMax(number: Int) -> (stringVal: String, origVal: Int) {`
	`    return(String(number), number)`
	`}`

• Optional Tuple Return Types	`func minimum(array: [Int]) -> (stringVal: String, origVal: Int)?`
	`{`
	`    if array.isEmpty { return nil }`
	`    else{`
	`        let value = array[0] < array[1] ? array[0]:array[1]`
	`        return(String(value), value)`
	`    }}`

Argument Labels
• Function parameters can be labelled.
• The argument label is used when calling the function
Parameter Names
• The parameter name is used in the implementation of the function.
• By default, parameters use their parameter name as their argument label.
	`func someFunction(labelName parameterName: Int) {`
	
	`}`
	
	`func addNumber(value1: Int, second value2: Int) {`
	`    // value1 -> bath label and parameter`
	`    // second -> label and value2->parameter`
	
	`}`
	
	`addNumber(value1: 2, second: 5)`

• Omitting Argument Labels	`func addTwoNumber(_ value1: Int, value2: Int) {`
	
	`}`
	`addTwoNumber(2, value2: 5)`

• Default Parameter Values	`func someFunc(withoutDefault: Int, withDefault: Int = 12)`
	`{`
	`}`
	`someFunction(withoutDefault: 3, withDefault: 6) //`
	`withDefault is 6`
	`someFunction(withoutDefault: 4) // withDefault is 12`

Function

- **Variadic Parameter**
- Accepts zero or more values of a specified type.
- Written by inserting three period characters (...) after the parameter's type name.

```
func arithmeticMean(_ numbers: Double...)
-> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
arithmeticMean(1, 2, 3, 4, 5, 6)
arithmeticMean(1, 8.25, 18.75)
```

- **In-Out Parameters**
- Function parameters are constants by default.
- To modify a parameter's value, and to keep those changes permanent define that parameter as an *in-out*.
- An in-out parameter has a value that is passed *in* to the function, is modified by the function, and is passed back *out* of the function to replace the original value
- **In-out parameters cannot have default values, and variadic parameters cannot be marked s inout.**

```
func swapTwoInts(_ a: inout Int, _ b:
inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
var someInt = 3, anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and
anotherInt is now \(anotherInt)")
// Prints "someInt is now 107, and
anotherInt is now 3“
```

Function

- Function Types
- Every function has a specific *function type*, made up of the parameter types and the return type of the function.

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}


/*The type of function is (Int, Int) -> Int. This can
be read as:
“A function that has two parameters, both of
type Int, and that returns a value of type Int.”*/
```

- Using Function Types
- function types can be used just like any other types in Swift.
- For example, a constant or variable can be defined to be of a function type and an appropriate function to that variable can be assigned:

```
var mathFunction: (Int, Int) -> Int = addTwoInts


/*This can be read as:
“Define a variable called mathFunction, which has a
type of ‘a function that takes two Int values, and
returns an Int value.’ Set this new variable to refer
to the function called addTwoInts.”*/
```

Function

- Function Types as Parameter Types
- A function type such as
- **(Int, Int) -> Int** can be used as a parameter type for another function.

```
func addTwoInts(val1: Int, val2 :Int)->Int{
    return(val1 + val2)
}
func printMathResult(_ mathFunction: (Int, Int) -> Int, a:
Int, b: Int) {
    print("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, a: 3, b: 5)
```

- Function Types as Return Types
- A function type can be used as the return type of another function.
- It's done by writing a complete function type immediately after the return arrow (->) of the returning function.

```
func stepForward(_ input: Int) -> Int { return input + 1 }
func stepBackward(_ input: Int) -> Int { return input - 1}


func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    return backward ? stepBackward : stepForward
}
```

Function

- Nested Functions
- Functions inside the bodies of other functions are known as *nested functions*.
- Nested functions are hidden from the outside world by default,
- It can still be called and used by their enclosing function.
- An enclosing function can also return one of its nested functions to allow the nested function to be used in another scope.

```swift
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }
    func stepBackward(input: Int) -> Int { return input - 1 }
    return backward ? stepBackward : stepForward
}

var currentValue = -4
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)

// moveNearerToZero now refers to the nested stepForward() function

while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}

print("zero!")
// -4...
// -3...
// -2...
// -1...
// zero!
```

Closures

Closures are self-contained blocks of functionality that can be passed around and used in code.
Closures in Swift are similar to blocks in C and to lambdas in other programming languages.
Closures take one of three forms:

- Global functions are closures that have a name and do not capture any values.
- Nested functions are closures that have a name and can capture values from their enclosing function.
- Closure expressions are unnamed closures written in a lightweight syntax that can capture values from their surrounding context.

```swift
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
func backward(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2
}
var reversedNames = names.sorted(by: backward)
// reversedNames is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

Closures

1.6.1 Closure Expressions

- *Closure expressions* are a way to write inline closures in a brief, focused syntax.

```
{ (parameters) -> return type in
    statements
}
```

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in return s1 > s2} )
```

For the inline closure expression, the parameters and return type are written *inside* the curly braces. The start of the closure's body is introduced by the *in* keyword which indicates that the definition of the closure's parameters and return type has finished, and the body of the closure is about to begin.

Closures

Inferring Type From Context

- The sorted(by:) method is being called on an array of strings, so its argument must be a function of type (String, String) -> Bool.
- This means that the (String, String) and Bool types do not need to be written as part of the closure expression's definition.

```
reversedNames = names.sorted(by: { s1, s2 in
return s1 > s2 } )

/*Because sorted is called on names, which is an
array of string hence mentioning datataype is
not required.*/
```

Implicit Returns from Single-Expression Closures

- Single-expression closures can implicitly return the result of their single expression by omitting the return keyword from their declaration, as in this version of the previous example:

```
reversedNames = names.sorted(by: { s1, s2 in s1
> s2 } )

/*Because the closure's body contains a single
expression (s1 > s2) that returns a Bool value,
 there is no ambiguity, and the return keyword
can be omitted.*/
```

Closures

Shorthand Argument Names

- Swift automatically provides shorthand argument names to inline closures, which can be used to refer to the values of the closure's arguments by the names **$0**, **$1**, **$2**, and so on.
- The number and type of the shorthand argument names will be inferred from the expected function type.
- The in keyword can also be omitted, because the closure expression is made up entirely of its body:

```
reversedNames = names.sorted(by: { $0 > $1 }
)


/*Here, $0 and $1 refer to the closure's
first and second String arguments.*/

/* The expression (between curly braces)
becomes the body hence in keyword is not
required*/
```

Operator Methods

- Swift's String type defines its string-specific implementation of the greater-than operator (>) as a method that has two parameters of type String, and returns a value of type Bool.
- Simply pass in the greater-than operator, and Swift will infer that it's string-specific implementation is required.

```
reversedNames = names.sorted(by: >)
```

Closures – Trailing Closures

- If a closure expression is passed to a function as it's final argument and the closure expression is long, it can be useful to write it as a *trailing closure* instead.

- A trailing closure is written after the function call's parentheses, even though it is still an argument to the function.

- In trailing closure syntax, argument label for the closure is not written as part of the function call.

- If a closure expression is function's or method's only argument and that expression is provided as a trailing closure, then parentheses **()** after the function or method's name is not required at function call:

```
let closureAdd = {
    (x: Int, y: Int)->Int in
    return x + y;
}

func takeClousreAsInput(_ input1:Int, _ input2:Int,
_ closure:(Int, Int)->Int){
    print(closure(input1, input2));
}

takeClousreAsInput(5, 6, closureAdd);
//passing closure expression


takeClousreAsInput(5, 6, {
    (x: Int, y: Int)->Int in
    return x+y;
})

takeClousreAsInput(23, 12){
    (x:Int, y:Int)->Int in
    return (x - y);
}
```

Closures – Capturing Values

- A closure can *capture* constants and variables from the surrounding context in which it is defined.
- A nested function is a simplest form of closure which can capture values.
- The closure can then refer to and modify the values of those constants and variables from within its body, even if the original scope that defined the constants and variables no longer exists. This is because **functions and closures are *reference types***.

```
func makeIncrementer(forIncrement increaseBy:Int)->()->Int{
    var total = 0;
    func incrementer()->Int{
        total += increaseBy;
        return total
    }
    return incrementer //return the reference
}
let incrementer = makeIncrementer(forIncrement: 10)
print(incrementer())
print(incrementer())


let incrementByTen = makeIncrementer(forIncrement: 10)
print(incrementByTen())  // returns a value of 10
print(incrementByTen())  // returns a value of 20
print(incrementByTen())  // returns a value of 30
```

Closures – Escaping Closures

- A closure is said to *escape* a function when the closure is passed as an argument to the function, but is called after the function returns.
- When a function takes a closure as one of its parameters, write **@escaping** before the parameter's type to indicate that the closure is allowed to escape.

```
var completionHandlers = [()->Void]();

func someFunctionWithEscapingClosure(_
completionHandler:@escaping()->Void)->Void{
    completionHandlers.append(completionHandler);
}

func someFunctionWithNonEscapingClosure(_ closure:()->Void)-
>Void{
    closure();
}

var x = 100;
someFunctionWithEscapingClosure { x = 200 }
someFunctionWithNonEscapingClosure { x = 300 }
print(x); //prints 300
completionHandlers.first?();
print(x); // prints 200
```

Closures – Escaping Closures

- Marking a closure with **@escaping** means **self** must be done explicitly within the closure

```
func someFunctionWithNonescapingClosure(closure: () -> Void)
{
    closure()
}

class SomeClass {
    var x = 10
    func doSomething() {
        someFunctionWithEscapingClosure { self.x = 100 }
        someFunctionWithNonEscapingClosure{ x = 200 }

    }
}

let instance = SomeClass()
instance.doSomething()
print(instance.x)  // Prints "200"
completionHandlers[1]()
print(instance.x) // Prints "100"
```

Closures - Autoclosures

- Automatically created to wrap an expression that's being passed as an argument to a function.
- It doesn't take any arguments.
- when it's called, it returns the value of the expression that's wrapped inside of it.
- It lets to omit braces around a function's parameter by writing a normal expression instead of an explicit closure.
- An autoclosure lets evaluation to be delayed because the code inside isn't run until the closure is called.

```
var cityNames = ["Auckland", "Welllington", "Dunedin",
"Christchurch"]

let cityClosure = { cityNames.remove(at: 0)};
//not removed until called

print(cityNames.count)
print("\(cityClosure())")
print(cityNames.count)

func serve(_ closure:()->String){
    print("\(cityClosure())")
}

serve(cityClosure)

func serve(_ closure:@autoclosure()->String){
    print("\(cityClosure())")
}

serve(cityNames.remove(at: 0))

print(cityNames.count)
```

Closures - Autoclosures

- To create an autoclosure that is allowed to escape, use both @autoclosure and @escaping attributes

```
var cityNames = ["Auckland", "Welllington", "Dunedin", "Christchurch"]
var cityNamesArray: [() -> String] = []


func addCityNames(_ cityName: @autoclosure @escaping () -> String) {
    cityNamesArray.append(cityName)
}
print(cityNames.count)

addCityNames(cityNames.remove(at: 0))
addCityNames(cityNames.remove(at: 0))

print("Collected \(cityNamesArray.count) closures.")
// Prints "Collected 2 closures."

for cityName in cityNamesArray {
    print("Now removing \(cityName())!")
}

// Prints "Now removing Auckland !"
// Prints "Now serving Wellington!"
print(cityNames.count)
```

Enumerations

- Assign related names to a set of integer values
- In swift, enumeration do not have to provide value for each case.
- Value for a case can be a string, character, integer or floating-point.
- Cases can also specify associated values.
- Common set of related cases can also be defined.
- Multiple cases can appear on a single line separated by commas

- *Can provide computed properties*
- *Can define initializers to provide initial case values*
- *Can be extended to expand their functionality*
- *Can confirm to protocols*

```
enum SomeEnumeration {
    // enumeration definition goes here
}
enum CompassPoint {    // Swift enumeration cases
    case north         // are not assigned a
    case south         // default integer value
    case east          // when they are created.
    case west
}


enum Planet {
    case mercury, venus, earth, mars,
    jupiter, saturn, uranus, neptune
}
```

Enumerations

- Each enumeration definition defines a brand new type.
- The type of **directionToHead** is inferred when it is initialized with one of the possible values of **CompassPoint**.
- Once **directionToHead** is declared as a **CompassPoint**, it’s value can be changed using a shorter dot syntax:
- Individual enumeration values can be matched with a **switch** statement:

```
var directionToHead = CompassPoint.west


directionToHead = .east
directionToHead = .south

switch directionToHead {
case .north:
    print("Lots of planets have a north")
case .south:
    print("Watch out for penguins")
case .east:
    print("Where the sun rises")
case .west:
    print("Where the skies are blue")
}// Prints "Watch out for penguins"
```

Enumerations – Associated Values

- Swift enumerations can store associated values of any given type.
- The value types can be different for each case of the enumeration if needed.

- The different value types can be checked using a switch statement, as before.
- This time, however, the associated values can be extracted as part of the switch statement.
- Each associated value can be extracted as a constant or a variable for use within the **switch** case's body:

```
enum Barcode {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
}
var productBarcode = Barcode.upc(8, 85909, 51226, 3)
productBarcode = .qrCode("ABCDEFGHIJKLMNOP")

switch productBarcode {
case .upc(let numberSystem, let manufacturer, let product,
let check):
    print("UPC: \(numberSystem), \(manufacturer),
\(product), \(check).")
case .qrCode(let productCode):
    print("QR code: \(productCode).")
}
// Prints "QR code: ABCDEFGHIJKLMNOP."
```

Enumerations – Associated Values

- If all of the associated values for an enumeration case are extracted as constants, or variables, then a single’ **var**’ or ’**let**’ annotation can be placed before the case name

```
switch productBarcode {
case let .upc(numberSystem, manufacturer, product, check):
    print("UPC : \(numberSystem), \(manufacturer), \(product), \(check).")
case let .qrCode(productCode):
    print("QR code: \(productCode).")
}
// Prints "QR code: ABCDEFGHIJKLMNOP."
```

Enumerations – Raw Values

- As an alternative to associated values, enumeration cases can come prepopulated with default values (called *raw values*), which are all of the same type.

- Implicitly Assigned Raw Values - Swift automatically assign values to the cases of the enumerations which store Integers or Strings.

- When strings are used for raw values, the implicit value for each case is the text of that case’s name.

```
enum ASCIIControlCharacter: Character {
    case tab = "\t"
    case lineFeed = "\n"
    case carriageReturn = "\r"
}
```

```
enum Planet: Int {
    case mercury = 1, venus, earth, mars,
jupiter, saturn, uranus, neptune
}
```

```
enum CompassPoint: String {
    case north, south, east, west
}
```

Enumerations – Raw Values

- Initializing from a Raw Value -

- If an enumeration is defined with a raw-value type, the enumeration automatically receives an initializer.
- This initializer takes a value of the raw value's type (as a parameter called **rawValue**) and returns either an enumeration case or **nil**.
- This initializer can be used to try to create a new instance of the enumeration.

```
let earthsOrder = Planet.earth.rawValue
// earthsOrder is 3

let sunsetDirection = CompassPoint.west.rawValue
// sunsetDirection is "west"

let possiblePlanet = Planet(rawValue: 7)
// possiblePlanet is of type Planet? and
// equals  Planet.uranus
```

Enumerations – Recursive Enumerations

- A *recursive enumeration* is an enumeration that has another instance of the enumeration as the associated value for one or more of the enumeration cases.
- An enumeration case is recursive by writing **indirect** before it, which tells the compiler to insert the necessary layer of indirection. For example,

```
enum Exm{
  case number(Int)
  indirect case addition(Exm, Exm)
  indirect case multiplication(Exm, Exm)
}
```

To enable indirection for all of the enumeration’s cases write **indirect** before the beginning of the enumeration

```
indirect enum Exm{
  case number(Int)
  case addition(Exm, Exm)
  case multiplication(Exm, Exm)
}
```

Recursive Enumeration - Example

```
indirect enum ArithmeticExpression {
    case number(Int)
    case addition(ArithmeticExpression,
ArithmeticExpression)
    case multiplication(ArithmeticExpression,
ArithmeticExpression)
}
```

```
//Solve (5 + 4) * 2
let five = ArithmeticExpression.number(5)
let four = ArithmeticExpression.number(4)
let sum = ArithmeticExpression.addition(five,
four)
let product =
ArithmeticExpression.multiplication(sum,
ArithmeticExpression.number(2))
```

```
func evaluate(_ expression:
ArithmeticExpression) -> Int {
    switch expression {
    case let .number(value):
        return value
    case let .addition(left, right):
        return evaluate(left) + evaluate(right)
    case let .multiplication(left, right):
        return evaluate(left) * evaluate(right)
    }
}
```

```
print(evaluate(product))
// Prints "18"
```

Classes and Structures

- In Swift, class or a structure is defined in a single file and the code is made available to any external interface to this class or structure.

Comparing Classes and Structures

Classes and structures have many things in common.
Both can:

- Define **properties** to store values
- Define **methods** to provide functionality
- Define **subscripts** to provide access to their values using subscript syntax
- Define **initializers** to set up their initial state
- Be **extended** to expand their functionality beyond a default implementation
- Conform to **protocols** to provide standard functionality of a certain kind

```swift
class SomeClass {
    // class definition goes here
}
struct SomeStructure {
   // structure definition goes here
}

struct Resolution {
    var width = 0
    var height = 0
}
class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}

/*The syntax for creating instances is
very similar for both structures and
classes:*/
let resolution = Resolution()
let videoMode = VideoMode()
```

Class - NOTE

Classes have additional capabilities that structures do not:

- Inheritance enables one class to inherit the characteristics of another.
- Type casting enables to check and interpret the type of a class instance at runtime.
- Deinitializers enable an instance of a class to free up any resources it has assigned.
- Reference counting allows more than one reference to a class instance.

Classes and Structures - Properties

2.1.1 Accessing Properties

- Access the properties of an instance using *dot syntax*.
- Use dot syntax to assign a new value to a variable property:

```
print("The width of resolution is \(resolution.width)")
// Prints "The width of resolution is 0“


print("The width of videoMode is \(videoMode.resolution.width)")
// Prints "The width of videoMode is 0"


videoMode.resolution.width = 1280
```

2.1.2 Structures and Enumerations Are Value Types

- A *value type* is a type whose value is *copied* when it is assigned to a variable or constant, or when it is passed to a function.

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
```

2.1.3 Classes Are Reference Types

- Unlike value types, *reference types* are *not* copied rather than a reference to the same existing instance is used instead.

```
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0


let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0
```

2.1.4 Identity Operators

To compare two instance of a class, Swift provides two identity operators:

Identical to (===) - Both class instances are same.

Not identical to (!==) - Both class instances Not Same

```
if tenEighty === alsoTenEighty {
    print("tenEighty and alsoTenEighty refer to 
the same VideoMode instance.")
}
// Prints "tenEighty and alsoTenEighty refer to
// the same VideoMode instance."
```

2.1.5 Choosing Between Classes and Structures

As a general guideline, consider creating a structure when one or more of these conditions apply:

- The structure's primary purpose is to encapsulate a few relatively simple data values.
- It is reasonable to expect that the encapsulated values will be copied rather than referenced when an instance of that structure is assigned or passed around.
- Any properties stored by the structure are themselves value types, which would also be expected to be copied rather than referenced.
- The structure does not need to inherit properties or behavior from another existing type.

Examples of good candidates for structures include:

- The size of a geometric shape, perhaps encapsulating a width property and a height property, both of type Double.
- A way to refer to ranges within a series, perhaps encapsulating a start property and a length property, both of type Int.
- A point in a 3D coordinate system, perhaps encapsulating x, y and z properties, each of type Double.

Classes and Structures

- In all other cases, define a class, and create instances of that class to be managed and passed by reference.

- In practice, this means that most custom data constructs should be classes, not structures.

- In Swift, many basic data types such as String, Array, and Dictionary are implemented as structures.

- This means that data such as strings, arrays, and dictionaries are copied when they are assigned to a new constant or variable, or when they are passed to a function or method.